9-29-2005

Efficient Self-Join Algorithm in Interval-based Temporal Data Models

Seo-Young Noh
Iowa State University

Shashi K. Gadia
Iowa State University, gadia@iastate.edu

Efficient Self-Join Algorithm in Interval-based Temporal Data Models

Abstract
Interval-based temporal data model is a popular data model in temporal databases. It uses time intervals for representing the period of validity of a tuple, leading to unavoidable self-joins when combining tuples for objects. It requires k+1-way self-join for k conjunctive conditions. Join operations are one of the most expensive operations in databases and they are even more serious in temporal databases because of growing data. There are many join algorithms for temporal databases. However, they focus on joining different inputs rather than an identical input, leading to multiple scans for the identical input. Advanced 2-way join algorithms avoid a quadratic disk I/O complexity, but they are affected by the number of self-joins and partition sizes. In this paper, we address the problem of self-joins in the interval-based temporal data model and introduce a stream-based self-join algorithm. The proposed algorithm shows that it achieves a single relation scan for k-way self-join and its performance is not affected by partition sizes.

Keywords
Self-Join Algorithm, Interval-based Data Model, Temporal Databases

Disciplines
Databases and Information Systems

Efficient Self-Join Algorithm in Interval-based Temporal Data Models*

Seo-Young Noh and Shashi K. Gadia

Technical Report #05-22
Department of Computer Science
Iowa State University
Ames, Iowa, USA

September 29, 2005

Abstract

Interval-based temporal data model is a popular data model in temporal databases. It uses time intervals for representing the period of validity of a tuple, leading to unavoidable self-joins when combining tuples for objects. It requires $k+1$-way self-join for k conjunctive conditions. Join operations are one of the most expensive operations in databases and they are even more serious in temporal databases because of growing data. There are many join algorithms for temporal databases. However, they focus on joining different inputs rather than an identical input, leading to multiple scans for the identical input. Advanced 2-way join algorithms avoid a quadratic disk I/O complexity, but they are affected by the number of self-joins and partition sizes. In this paper, we address the problem of self-joins in the interval-based temporal data model and introduce a stream-based self-join algorithm. The proposed algorithm shows that it achieves a single relation scan for k-way self-join and its performance is not affected by partition sizes.

Keywords: Self-Join Algorithm, Interval-based Data Model, Temporal Databases

*This work has been sponsored in part by Information Infrastructure Institute and a grant from Baker Foundation Trust at Iowa State University.

1 Introduction

Temporal databases are capable of storing and querying history of objects or events. Researchers have developed and introduced many different types of temporal data models. Temporal data models are relatively complicated because they should capture time-aspect data characteristics. In general, temporal data models can be broadly categorized based on domain representation and timestamping. Domain representation indicates how to represent the valid time domain of objects while timestamping indicates how to assign domain representation to objects.

In this paper, we consider an interval-based temporal data model which is a popular data model in temporal databases. This data model uses time intervals for domain representation and tuple-level timestamping. One advantage of this data model is its fast implementation on top of conventional relational databases, availing well-developed optimizer.

In the interval-based data model, an object is modeled in multiple tuples in a relation because an event is validated with an interval. When we need to retrieve qualified objects, all tuples for an object should be combined and evaluated. Combining tuples for an object leads to self-joins and it is unavoidable when a conjunction is evaluated on a single tuple in the object. For example, suppose that we want to retrieve all employees who have experience in Hardware *and* Software department. Since a tuple has a single value for time interval for an attribute, the department attribute value cannot be Hardware and Software simultaneously. Therefore we needs 2-way join for the employee relation. In general, if there are k conjunctive conditions in queries, it requires $k+1$-way join in the interval-based data model [2]. We call a join for an identical relation *self-join* which is a special type of general joins. In a self-join an object is computed with itself.

Join operations are one of the most expensive operations in databases and they are even more expensive in temporal databases because temporal databases are accumulating histories and hence larger. We can find many research papers on join operations for temporal data, but it is hard to find treatments of self-joins. General temporal join algorithms require multiple scans for k-way self-join and performance is affected by object's partition sizes. In this paper, we introduce a stream-based self-join algorithm which requires only a single scan for k-way self-join and is not affected by the number of self-joins and partition sizes.

The organization of the rest of this paper is as follows. Section 2 discusses related work for join algorithms. Section 3 introduces the interval-based temporal data model and its hypothetical query language. Section 4 discusses three join algorithms. Section 5 introduces the stream-based self-join algorithm. Section 6 compares the disk I/O complexities of four algorithms for self-joins and discusses the performance expectations. Section 7 concludes our work.

2 Related Work

In conventional databases, join algorithms are used for combining tuples from relations based on a join condition. There are many different types of algorithms from simple nested-loop joins to index-based joins [5]. Join operations in temporal databases have the same concept as that of the conventional databases. However, they are more complicated because of temporal data characteristics.

Temporal join operations can be found in literature. There are two main categories such

as temporal join algorithms with indices and join algorithms without indices.

Zhang et al. [13] introduced temporal join algorithms using indices. They proposed temporal index-based join algorithms as well as various optimization techniques that further improve join performance. Enderle et al. [1] introduced algorithms that join interval data in relational databases. The join algorithms use an index called Relational Interval Tree. Based on the Relational Interval Tree, the join algorithms are implemented on top of relational databases. Son and Elmasri [9] introduced a temporal join algorithm with Time Index. Time Index determines the exact partitioning intervals that make each partition fit into the assigned buffer space so that each partition can be performed without additional disk accesses once in memory.

Join algorithms in the temporal databases can be extended from join algorithms in conventional databases without using indices. Gao et al. [3] summarized join algorithms in temporal databases including nested-loop-based join, and sort-merge-based join. Soo et al. [10] introduced a partition-based valid time natural join algorithm, achieving linear ordered I/O complexity. These algorithms are all implemented based on relation scans without indexes.

Despite much research work on temporal joins, it is hard to find methodologies for self-joins. As defined by Mishra and Eich [5], joins are generally considered as combining tuples from *two different relations* based on some common information. In the interval-based data model, self-joins appear in temporal queries when combining tuples for an object and it may degrade system performances if we use general temporal join algorithms for self-joins.

3 Interval-based Models

In this section, we will discuss the general concept of the interval-based temporal data model and a hypothetical query language for the data model.

3.1 Overview

We consider an interval-based data model that uses time-intervals and tuple-level time stamping, which is the most popular approach for interval-based data models. It is called *bitemporal data model* if the data model manages transaction time and valid time of tuples [11]. In this paper, we only consider one dimensional temporal data model. In the interval-based temporal data model, in addition to usual attributes, Start and End attributes are used to specify the period of validity for the information in the tuple [6, 8].

Figure 1 shows `Emp` temporal relation that maintains the history of employees with name, salary, and department information. Every tuple has Start and End attributes indicating the period of validity of the tuple. In `Emp` relation, an object consists of multiple tuples, where each tuple represents an event. For example, two tuples, ⟨Tom, 5000, Hardware, [41,51]⟩ and ⟨Tom, 5000, Software, [52,60]⟩, are distinct events and the latter tuple was created when Tom moved to Software department from Hardware department. Therefore, there exists no tuples whose time intervals overlap.

3.2 Interval-based Structured Query Langauge

We define a hypothetical query language, ISQL (Interval-based Structured Query Language). ISQL is a query language for the interval-based data model and is used to address self-join problems in the interval-based data model. A simplified BNF form of ISQL is as follows:

Name	Salary	DName	Start	End
Tom	45000	Sales	0	20
Tom	50000	Hardware	41	51
Tom	50000	Software	52	60
Tom	60000	R&D	61	70
Jane	50000	Software	10	54
Jane	55000	R&D	55	60
Jane	60000	R&D	61	70

Figure 1: Emp relation

ISQL := SELECT <attribute list>
[RESTRICTED TO <time interval or time instant>]
FROM <relation list>
[WHERE <boolean expression>]

ISQL is very similar to classical SQL except `RESTRICTED TO` clause. `RESTRICTED TO` clause is to capture specific time domain information of qualified tuples evaluated by a boolean expression in `WHERE` clause.

For illustration of self-join problems, consider the following query:

Query: Give current departments of employees who have experience in Hardware, Software, Sales, and R&D departments.

```
SELECT E1.DName
RESTRICTED TO NOW
FROM Emp E1, Emp E2, Emp E3, Emp E4
WHERE E1.Name = E2.Name
      AND E2.Name = E3.Name
      AND E3.Name = E4.Name
      AND E1.DName='Hardware'
      AND E2.DName='Software'
      AND E3.DName='Sales'
      AND E4.DName='R&D'
```

As we can see in this example, the conditions have three conjunctions, requiring 4-way self-join. In the interval-based data model, this self-join is unavoidable because tuples for an object are scattered in a relation.

4 Join Algorithms

4.1 Block Nested Loop Join

The simplest and most directive join algorithm is the nested-loop join. One of derivations of nested loop joins is *block nested loop join* to utilize a buffer pool. Block nested loop join algorithms break the outer relation (r) into blocks that can fit into the available buffer pages and scanning all of the inner relation (s) for each block of the outer relation [7].

The temporal block nested loop join can be constructed from the conventional block nested loop join as making the time stamp predicate be evaluated at the same time as the predicate on the attributes [3].

Algorithm 1 shows temporal block-nested loop join[1]. In the algorithm, we denote condition and timestamp as C and T, respectively. A joined tuple of tuple $x \in r$ and $y \in s$ is denoted as z, where attribute A represents all attributes of x except joining attribute C and attribute B represents all attributes of y except the joining attribute C. Function Overlap finds a maximum interval between two timestamps from tuple x and y. This algorithm is simple, but has a quadratic disk I/O complexity.

Algorithm 1 Block Nested Loop Join

```
1:  procedure BlockNestedLoopJoin(r, s, C)        ▷ r, s: input relations, C: condition
2:      for each block b_r ∈ r do
3:          for each block b_s ∈ s do
4:              for each tuple x ∈ b_r do
5:                  for each tuple y ∈ b_s do
6:                      if x[C] = y[C] and Overlap(x[T], y[T]) ≠ ∅ then
7:                          z[A] ← x[A]
8:                          z[B] ← y[B]
9:                          z[C] ← x[C]
10:                         z[T] ← Overlap(x[T], y[T])
11:                         result ← result ∪ {z}
12:                     end if
13:                 end for
14:             end for
15:         end for
16:     end for
17: end procedure
```

4.2 Sort Merge-based Join

We can also adapt conventional sort merge-based join algorithm to temporal databases. Sort merge-based join algorithms sort r and s on the join attributes and merge the two inputs from the relations. In order to sort two relations, we can use an external sorting algorithm. In merging step, we scan the relation r and s, looking for qualifying tuples. The two scans

[1] Algorithms introduced in this paper are modified versions of algorithms introduced by Gao et al. [3]

start at the first tuple in each relation. We advance the scan of r as long as current tuple $x \in r$ is less than current tuple $y \in s$. Similarly, we then advance the scan of s as long as current tuple $y \in s$ is less than current tuple $x \in r$ [7]. Algorithm 2 shows a temporal sort merge-based join algorithm.

Algorithm 2 Sort Merge-based Join

```
1:  procedure SortMergeBasedJoin(r, s, C)        ▷ r, s: input relations, C: condition
2:      r ← Sort(r, C)                            ▷ sorting using an external sorting algorithm
3:      s ← Sort(s, C)
4:      x ← NextTuple(x)                          ▷ the first tuple from r
5:      y ← NextTuple(y)                          ▷ the first tuple from s
6:      while x ≠ null and y ≠ null do
7:          while x[C] < y[C] do                  ▷ advance tuple x
8:              x ← NextTuple(x)                  ▷ next tuple of x
9:          end while
10:         while y[C] > y[C] do                  ▷ advance tuple y
11:             y ← NextTuple(y)                  ▷ next tuple of y
12:         end while
13:         k ← y                                 ▷ k is a pointer to y
14:         while x[C] = y[C] do
15:             y ← k                             ▷ reset y to its first tuple
16:             while x[C] = y[C] do
17:                 if Overlap(x[T], y[T]) ≠ ∅ then
18:                     z[A] ← x[A]
19:                     z[B] ← y[B]
20:                     z[C] ← x[C]
21:                     z[T] ← Overlap(x[T], y[T])
22:                     result ← result ∪ {z}
23:                 end if
24:                 y ← NextTuple(y)
25:             end while
26:             x ← NextTuple(s)
27:         end while
28:     end while
29: end procedure
```

In the sort merge-based join, line 13 assigns k to the first tuple of s such that $x[C] = y[C]$ because the tuples should be reused for the next tuple of x. Line 15 reassigns y to the first tuple of s which satisfies $x[C] = y[C]$.

4.3 Partition-based Join

Soo et al. [10] introduced a linear ordered valid-time natural join algorithm called partition-based join algorithm. This algorithm partitions relation r and s into n partitions. The join $r \bowtie s$ is computed by unioning the joins $r_i \bowtie s_i$, where $1 \leq i \leq n$.

When partitioning input relations, we can use a hash function, achieving a linear ordered I/O complexity. The advantage of partition-based join avoids the quadratic cost of nested loop evaluation and sorting. However, we must note that this is true under the ideal case such that partitions are small enough to fit in a buffer pool. For example, if the buffer pool size is smaller than a partition size, entire partition r_i cannot be loaded in the buffer. Temporal data is so accumulative that the partition size can exceed the buffer size. Partition-based algorithms are known that it shows superior performance when the sizes of input relations are different [3, 4]. Algorithm 3 shows a partition-based join algorithm.

Algorithm 3 Partition-based Join

```
1: procedure PartitionBasedJoin((r, s, C))        ▷ r, s: input relations, C: condition
2:     r ← Partition(r, C)                        ▷ partitioning using a hash function
3:     s ← Partition(s, C)
4:     for each p_r ∈ r and p_s ∈ s do
5:         for each tuple x ∈ p_r do
6:             for each tuple y ∈ p_s do
7:                 if x[C] = y[C] and Overlap(x[T], y[T]) ≠ ∅ then
8:                     z[A] ← x[A]
9:                     z[B] ← y[B]
10:                    z[C] ← x[C]
11:                    z[T] ← Overlap(x[T], y[T])
12:                    result ← result ∪ {z}
13:                end if
14:            end for
15:        end for
16:    end for
17: end procedure
```

5 Stream-based Self-Join Algorithm

Among the three join algorithms introduced in the previous section, the partition-based join algorithm for self-join seems to have the best I/O complexity. However, we must note that the algorithm expects that the input relations be different, leading to duplicate scanning for an identical relation in processing self-joins. More importantly, the algorithm can achieve linear ordered disk I/O complexity only when the partitions are small enough to fit in the buffer pool.

In this section, we introduce a stream-based self-join algorithm. The proposed self-join algorithm consists of three steps. First, like the partition-based join, it partitions the input relation based on objects. Second, it constructs a condition table for join conditions. Last, it reads tuples from partitions in a single stream and evaluates the condition table.

In our algorithm, we partition the input relation based on self-join attribute, grouping tuples based on objects so that each partition contains only related tuples for an object. Since we consider the interval-based temporal data model, self-joins combine all related tuples for

objects whose tuples are scattered in a relation.

A partition may consist of multiple blocks, and these blocks are clustered. Figure 2 shows a logical view of a cluster. We must note that blocks in a cluster do not need to be sequenced.

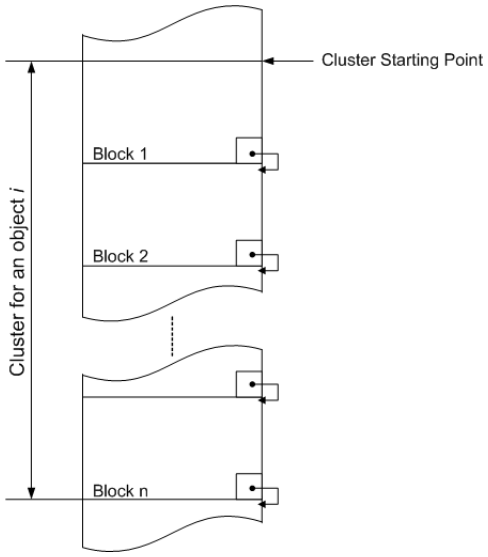

Figure 2: A logical view of a cluster

Algorithm 4 Stream-based Self-Join Algorithm

```
 1: procedure StreamBasedSelfJoin(r, C)          ▷ r: an input relation, C: condition
 2:     r ← Partition(s, C)
 3:     ConTable ← ConditionTable(C)
 4:     r[1..k] ← CreatePointers(C)
 5:     for each p_r ∈ r do
 6:         temp ← p_r
 7:         Initialize(ConTable, r[1..k])
 8:         for each x ∈ p_r do
 9:             r[1..k] ← x
10:             ConTable ← Update(ConTable, r[1..k])
11:             if ConTable = true then
12:                 result ← ∪{temp}
13:             end if
14:         end for
15:     end for
16: end procedure
```

Algorithm 4 shows the stream-based self-join algorithm. In the stream-based self-join algorithm, line 4 creates tuple pointers. These pointers represent relations shown in `FROM` clause in a query. Line 6 temporally saves a partition pointer to *temp* variable which should

be returned when the partition satisfies join conditions. Line 7 initializes the condition table and tuple pointers for a new partition. For each tuple from partition $p_r \in r$, line 9 through 11 updates the tuple pointers and the condition table, and checks if all condition items in the condition table have been set to true. Once the condition table is set to true, the partition is satisfied with the join conditions and should be returned.

It is worth noting that when updating the condition table, if a condition item has been already set to true, then the condition item is not evaluated any more. Another important aspect of proposed stream-based self-join algorithm does not have any timestamp comparisons. It is because the self-joins in the interval-based model are used for retrieving all tuples for an object. Since in this paper we assume that the temporal databases are sorted by time sequence, there are no two tuples whose timestamps are overlapped in the same object. This aspect implies that the stream-based self-join algorithm can be used in the conventional databases' self-join operations.

We must note that in this algorithm we create k pointers to implement a k-way self-join. These pointers move forward, pointing to a tuple in a single buffer. Since the algorithm moves tuple pointers in a buffer, it does not require any additional relation scans for k-way self-join.

For illustration of this algorithm, consider the query introduced in Section 3.2. The query requires 4-way self-join. Figure 3 describes how the stream-based self-join algorithm avoids multiple scans for the 4-way self-join.

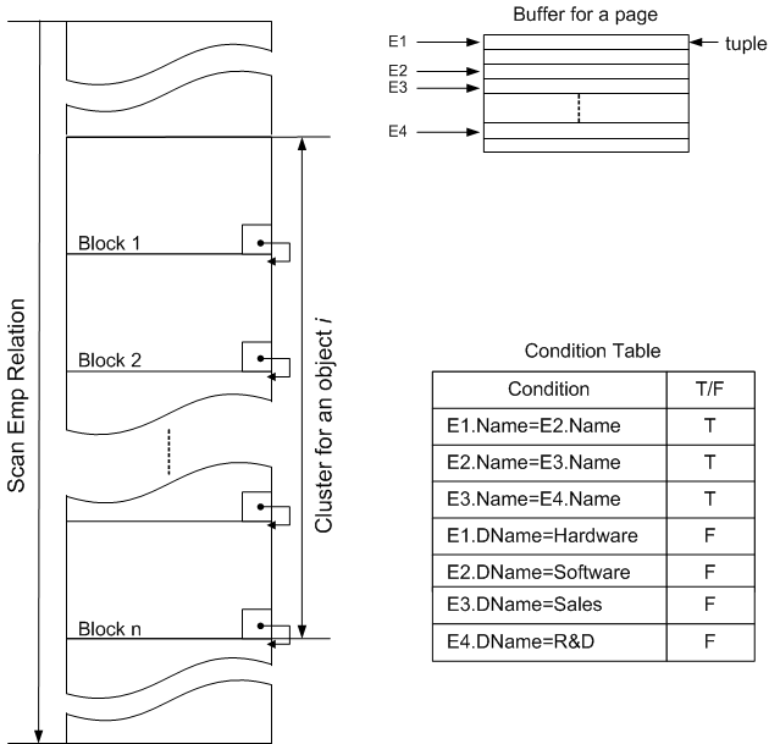

Figure 3: An example of special treatment for 3-way self-join

Each page in `Emp` relation is allocated to a frame in a buffer pool when it is requested. Since it is 4-way join, there are 4 pointers that point to tuples in the buffer. Each pointer moves forward while the condition table checks boolean conditions related to the pointer.

Each pointer stops if its corresponding boolean conditions are qualified. Whenever boolean values of predicates in the condition table are changed, the predicate P for `WHERE` clause is evaluated. Once P is set to true, tuples for the object are returned in streaming fashion. By using this stream-based self-join, we can implement self-join operation as a single relation scan, reducing tremendous number of disk block accesses.

6 Performance Evaluation

6.1 I/O Complexity Comparison for Self-Join

In order to compare disk I/O complexity, we will define some notations as shown in Table 1.

Notation	Meaning
r	input relation, or size of the relation
B	buffer pool, or size of the buffer pool
b	single buffer, or size of the single buffer
p	partition for an object, or size of the partition
n	number of buffers, e.g. B/b

Table 1: Notations for I/O complexity

Size is one important characteristics in temporal databases. It is more realistic to measure the disk I/O complexity in terms of the size of the input relation and the size of the buffer pool. Therefore, we will determine the complexities of the four algorithms for self-joins in terms of r, B, and p. In the following subsections, we consider 2-way self-join.

6.1.1 Block Nested Loop Join

In the block-nested loop join algorithm, it assigns $n-2$ blocks to an outer relation. For sake of simplicity, we assume that $n-2 \approx n$ and we do not consider the cost of output after self-join operations.

The algorithm first scans the outer relation. For each block in the inner relation, it eventually joins the block and r/B blocks in the outer relation because r/B blocks are already scanned in the buffer pool. Therefore, the I/O complexity of this algorithm can be driven as shown in Eq. 1

$$\begin{aligned} C_{BNLP} &= \frac{r}{b} + \frac{\frac{r}{b}}{n} \times \frac{r}{b}, \qquad \text{where } \frac{r}{b} = \#\text{ of blocks of } r \\ &= n \cdot \left(\frac{r}{B}\right) + n^2 \cdot \left(\frac{r}{B}\right)^2 \end{aligned} \tag{1}$$

In this complexity, we must note that the nested loop-join remains the same I/O complexity regardless of partition sizes.

6.1.2 Sort Merge-based Join

In sort merge-based join algorithm, we first sort input relations. Since we only dealing with self-joins, only one sort operation is enough for sorting the input relation. If we uses an external sorting algorithm, we can sort the input relation in $O(\frac{r}{b} log_n^{\frac{r}{b}})$. For joining the identical relation, we need to consider two cases: 1) partition sizes are smaller than the buffer pool size; 2) partition sizes are greater than the buffer pool size. Eq. 2 and Eq. 3 show the disk I/O complexities for the two cases, respectively.

$$\begin{aligned} C_{SMBJ} &= \frac{r}{b} \cdot log_n^{\frac{r}{b}} + \left(\frac{r}{b} + \frac{r}{b}\right), \qquad \text{if } p \leq B \\ &= n \cdot \frac{r}{B}\left(1 + log_n^{\frac{r}{B}}\right) + 2n \cdot \frac{r}{B} \\ &= n \cdot \frac{r}{B}\left(3 + log_n^{\frac{r}{B}}\right) \end{aligned} \tag{2}$$

$$\begin{aligned} C_{SMBJ} &= \frac{r}{b} \cdot log_n^{\frac{r}{b}} + \left(\frac{r}{b} + \frac{r}{b} \times \frac{p}{B}\right), \qquad \text{if } p > B \\ &= n \cdot \frac{r}{B}\left(1 + log_n^{\frac{r}{B}}\right) + n \cdot \frac{r}{B}\left(1 + \frac{p}{B}\right) \\ &= n \cdot \frac{r}{B}\left(2 + log_n^{\frac{r}{B}} + \frac{p}{B}\right) \end{aligned} \tag{3}$$

In the case of $p > B$, for each partition p_i, the partition should be self-joined, meaning that for each block in p_i, $\frac{p}{B}$ times disk accesses are required.

6.1.3 Partition-based Join

In the partition-based join algorithm, it first partitions the input relations and uses a hash function so that only linear scan is enough to partition the input relation. Since it needs to scan and hash (or write) the input relation, the partition process requires $2 \times \frac{r}{b}$ block accesses. Similarly to the sort merge-based join, the I/O complexity of partition-based join algorithm is affected by the partition sizes. Therefore, we need to consider two cases, respectively. Eq. 4 and Eq. 5 show the I/O complexities for the two cases.

$$\begin{aligned} C_{PBJ} &= 2 \cdot \frac{r}{b} + \left(\frac{r}{b} + \frac{r}{b}\right), \qquad \text{if } p \leq B \\ &= 4n \cdot \frac{r}{B} \end{aligned} \tag{4}$$

$$\begin{aligned} C_{PBJ} &= 2 \cdot \frac{r}{b} + \left(\frac{r}{b} + \frac{r}{b} \times \frac{p}{B}\right), \qquad \text{if } p > B \\ &= n \cdot \frac{r}{B}\left(3 + \frac{p}{B}\right) \end{aligned} \tag{5}$$

6.1.4 Stream-based Self-Join

As we discussed in Section 5, the stream-based self-join partitions the input relation. To join k-way self-join, it requires only one pass scan regardless of the partition sizes. Therefore, the disk I/O complexity of this algorithm can be driven as shown in Eq. 6.

$$
\begin{aligned}
C_{SBSJ} &= 2 \cdot \frac{r}{b} + \frac{r}{b} \\
&= 3n \cdot \frac{r}{B}
\end{aligned} \tag{6}
$$

There are two important differences between the partition-based join algorithm and the proposed self-join algorithm. As we have already noted unlike the partition-based join algorithm, the proposed algorithm is not affected by the partition sizes. The more importantly, the proposed algorithm shows the same disk I/O complexity for k-way self-join.

6.2 Performance Expectation

Based on disk I/O complexities discussed in the previous section, we can estimate general performance of the four join algorithms for self-joins. To make comparison simple, we exclude output relations and partition size considerations. We assume that k-way self-join is $k-1$ times of 2-way self-join.

Figure 4 shows disk block access comparisons for four algorithms. In this comparison, disk block accesses are measured for 2-way self-join in terms of r/B. As shown in Figure 4, the block nested loop join shows the worst performance and the other algorithms show the similar performance. The stream-based self-join algorithm shows the best performance result as the input relation size is growing.

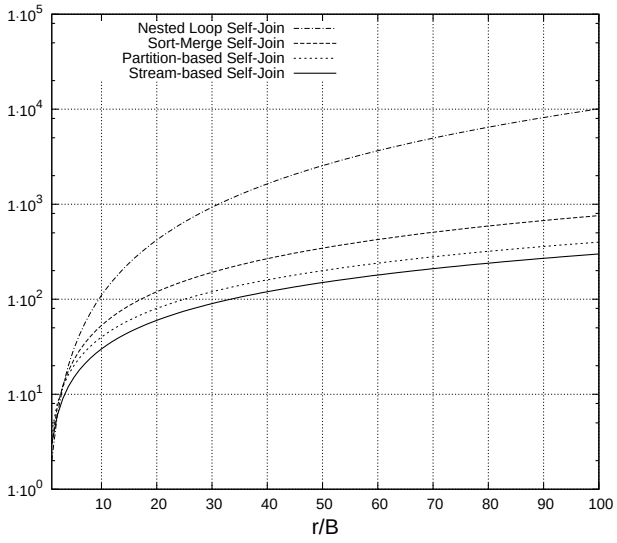

Figure 4: Block access comparison for 2-way self-join (logarithmic scale)

Figure 5 shows disk block access comparisons for k-way self-join. The algorithms except the stream-based self-join algorithm increase the number of disk access as the number of self-joins increases. However, the stream-based self-join shows the same disk block accesses regardless of the number of self-joins.

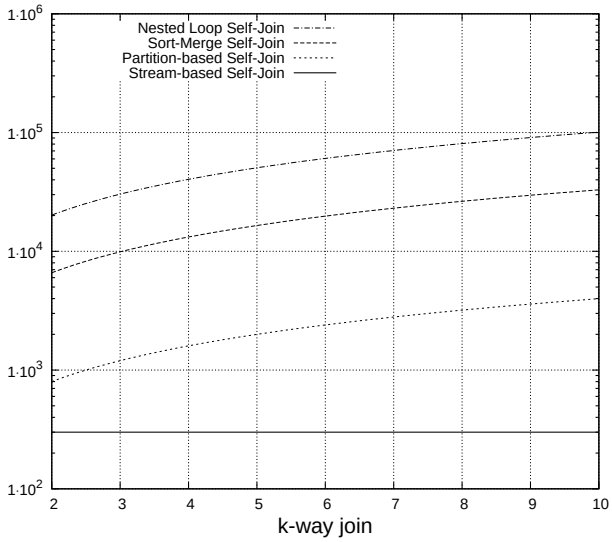

Figure 5: Block access comparison for k-way self-join (logarithmic scale)

7 Conclusion

Join operations are one of the most expensive operations in conventional databases and they pose even more serious problems in temporal databases. In the interval-based temporal data model, self-joins are unavoidable for combining tuples for an object. It requires $(k+1)$-way self-join for k conjunctive conditions.

There are many different join algorithms in temporal databases. Some of them extends conventional join algorithms and some use index mechanism. We have discussed three join algorithms without index structures whose complexities are from a polynomial to a linear. Among the join algorithms, the partition-based join algorithm shows the best disk I/O complexity, avoiding a quadratic nested evaluation and sorting. However, as we noted, the I/O complexity cannot be guaranteed when partition sizes exceed the size of a buffer pool. The three algorithms need multiple scans for processing k-way self-join.

In this paper, we have introduced a stream-based self-join algorithm. The stream-based self-join algorithm is based on the two observations. First we can construct a relation by partitioning tuples that represent an object. Second, there is only one-to-one mapping between an object and a partition, saying that an object p_i is not related to an object p_j if $i \neq j$. Therefore, we can partition an input relation by objects. These observations make it possible for us to remedy self-join problems in the interval-based data model, leading that the algorithm is not affected by partition sizes and the number of self-joins. The proposed algorithm shows just one pass scanning is enough for k-way self-join and its performance remains the same regardless of partition sizes.

Despite tremendous research work on general join algorithms in temporal databases, not much attention has been paid on self-join problems in temporal databases. Since data amassed over time is rapidly growing, joining an identical relation multiple times degrades system performances significantly. This paper provides a simple, yet elegant approach to resolve the self-join problems in the most popular temporal data model. We hope that our proposed algorithm provides a valuable insight on the implementation of self-join operations in temporal

databases.

References

[1] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 683–694, Paris, France, 2004.

[2] S. K. Gadia and S. S. Nair. Temporal databases: A prelude to parametric data. In *[12]*, pages 28–66. 1993.

[3] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[4] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[5] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[6] S. B. Navathe and R. Ahmed. Temporal extensions to the relational model and SQL. In *[12]*, pages 92–109. 1993.

[7] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 1999.

[8] R. T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[9] D. Son and R. Elmasri. Efficient temporal join processing using time index. In *Proceedings of the Eighth International Conference on Scientific and Statistical Database Management*, pages 252–261, Stockholm, Sweden, 1996.

[10] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 282–292, Houston, Texas, USA, 1994.

[11] A. U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A glossary of temporal database concepts. In *[12]*, pages 92–109. 1993.

[12] A. U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

[13] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Proceedings of the 18th International Conference on Data Engineering*, pages 103–113, San Jose, CA, USA, 2002.